

# Optimistic Active Replication\*

Pascal Felber<sup>†</sup>  
Bell Laboratories  
Lucent Technologies

André Schiper<sup>‡</sup>  
Swiss Federal Institute of  
Technology, Lausanne (EPFL)

## Abstract

*Replication is a powerful technique for increasing availability of a distributed service. Algorithms for replicating distributed services do however face a dilemma: they should be (1) efficient (low latency), while (2) ensuring consistency of the replicas, which are two contradictory goals. The paper concentrates on active replication, where all the replicas handle the clients' requests. Active replication is usually implemented using the Atomic Broadcast primitive. To be efficient, some Atomic Broadcast algorithms deliberately sacrifice consistency, if inconsistency is likely to occur with a low probability. We present in the paper an algorithm that handles replication efficiently in most scenarios, while preventing inconsistencies. The originality of the algorithm is to take the client-server interaction into account, while traditional solutions consider Atomic Broadcast as a black box.*

## 1 Introduction

Replication is a widely used technique for providing high-availability and fault-tolerance of critical services. However, developing replicated services is a challenging task. A replicated service must appear as a single highly-available logical entity to its client, which specifically means that the different copies must remain synchronized and consistent with each other. *Client* processes send requests to the *service* and wait for a reply. In the so-called *active* replication technique the client request is sent to all the server replicas using an Atomic Broadcast primitive (also called Total Order Broadcast), which ensures that the requests are delivered in the same order by all replicas. With active replication, every replica handles the request and sends back the reply to the client. While Atomic Broadcast preserve the consistency of a replicated service, it is considered costly to implement. A good survey of Atomic Broadcast algorithms can be found in [Déf00]. One of the key trade-off in Atomic Broadcast algorithms is between consistency and latency: to preserve consistency, the delivery of a message must be delayed until all replicas have agreed on its ordering, thus increasing the latency. To reduce the latency, some algorithms (e.g., [BSS91, KT91]) have sacrificed consistency, i.e., in some failure scenarios they may lead to the violation of the total order delivery of messages and thus to inconsistencies in the state of the servers. However, the fact that the servers are not always consistent with each other is not a problem in itself, as long as (1) inconsistencies can be repaired, and (2) they do not propagate to the clients. This requires however integration of the Atomic Broadcast algorithm with the client-server interaction schema, i.e., Atomic Broadcast is no more considered as a black box.

---

\*Submitted for publication.

<sup>†</sup>600 Mountain Avenue, Room 2B-303, Murray Hill, NJ 07974, USA. Tel: +1 (908) 582-0465. Fax: +1 (908) 582-1239. E-mail: pascal@research.bell-labs.com.

<sup>‡</sup>Operating Systems Laboratory, 1015 Lausanne, Switzerland. Tel: +41 (21) 693-4248. Fax: +41 (21) 693-6770. E-mail: andre.schiper@epfl.ch.

The paper presents an active replication algorithm, inspired by the Atomic Broadcast algorithm of [BSS91, KT91], which has a low latency and ensures that inconsistencies do not propagate to clients and can be repaired. The algorithm is optimistic in the sense that it assumes that failures are rare and is optimized for this case.

The rest of the paper is organized as follows. Section 2 introduces background concepts and related work about replication and optimistic algorithms. Section 3 describes the system model. Section 4 presents an overview of the optimistic active replication algorithm, and Section 5 gives a formal description of that algorithm. Finally, Section 6 concludes the paper. The complete proofs of the optimistic active replication algorithm are in Appendix A

## 2 Background and related work

### 2.1 Replication techniques

Fault-tolerance in distributed systems is typically achieved through replication. The literature distinguishes between two main classes of replication techniques: passive replication and active replication [GS97]. In passive replication the client only interacts with one replica, called the *primary*: the primary handles the client request and sends back the response. The primary also issues messages to the secondaries (the other replicas) in order to update their state. In active replication the client sends its request to all the replicas, which all handle the request and send back the response to the client. The client waits only for the first reply. Note that active replication requires the servers to be deterministic.

Ensuring consistency of the replicas is the main difficulty of replication techniques. One well known technique are *quorum systems* [Gif79]. However, quorum systems typically require a transactional infrastructure. This is not the case for group communication, another infrastructure for managing replication, which we consider here.

With active replication, consistency is ensured by having the clients invoke a group communication primitive called *Atomic Broadcast* (also called *Total Order Broadcast*) [HT93]. Atomic Broadcast guarantees that the requests sent by the clients are received by all replicas in the same order. With passive replication, consistency requires a group communication infrastructure that provides a *group membership service* (to select the primary), and a *view synchronous broadcast* (to be used by the primary to update the state of the secondaries) [GS97].

### 2.2 Design issues for atomic broadcast algorithms

We consider in the paper only active replication and Atomic Broadcast. Numerous Atomic Broadcast algorithms have been published in the last 15 years. A good survey can be found in [Déf00]. The different algorithms differ mainly by the assumptions they make with respect to the system model: they typically assume either a synchronous system, or an asynchronous system augmented with failure detectors. From a practical point of view, modeling the system as synchronous when the network and processor load are variable requires to be pessimistic for the bounds on message transmission delay and relative processor speeds. This leads to a large crash detection time, i.e., a large fail-over time, which is inadequate for time critical applications.

A better approach consists in assuming an asynchronous model augmented with a failure detector, which makes Atomic Broadcast solvable [CT96]. In this context, Atomic Broadcast algorithms can further be classified in two categories: (1) those that rely on a group membership oracle,<sup>1</sup> and (2) those that rely on a failure detector oracle. The Isis Atomic Broadcast algorithm of [BSS91]

---

<sup>1</sup>Also called group membership *service*.

typically are in the first category. The Chandra-Toueg Atomic Broadcast algorithm [CT96] is in the second category. As argued in [DSS98], the algorithms in the second category incur a smaller overhead when short fail-over time is required (e.g., time-critical applications). The algorithms in the second category also require a much simpler infrastructure. For this reason we consider in the paper an optimistic active replication technique that does not rely on a group membership oracle.

## 2.3 Optimistic algorithms

Trying to reduce the overhead related to short fail-over time is only a recent concern in the context of replication techniques based on group communication [DSS98]. For many years the concern was only to achieve fault-tolerance at low cost in the absence of failures. Achieving low cost in the absence of failures is of course extremely important, considering that fault-tolerance is usually considered to be expensive and to come with a significant overhead. The same comment applies to Atomic Broadcast, which was often criticized as being too expensive. Despite of that, designing *optimistic* Atomic Broadcast algorithms or *optimistic* active replication techniques was largely ignored until recently [PS98], even though optimistic algorithms were known since several years in the context of concurrency control [BHG87] and file system replication [RG93].

In the context of active replication, Pedone distinguishes two dimensions of optimism [Ped99]:

- i) Optimism at the level of the Atomic Broadcast algorithm.
- ii) Optimism at the level of the treatment of the client request by the replicated service.

The Optimistic Atomic Broadcast algorithm [PS98] is an example of (i). The algorithm makes the optimistic assumption that in a LAN messages are spontaneously received in total order with high probability, which is experimentally confirmed [PS98]. If this assumption is met, the algorithm delivers messages faster than known Atomic Broadcast algorithms. However, if the assumption does not hold, the algorithm is less efficient than other algorithms (but still delivers messages in total order).

The optimistic processing of transactions over Atomic Broadcast [KPAS99] is an example of (ii). [KPAS99] distinguishes two delivery events following the Atomic Broadcast of message  $m$ : the optimistic delivery denoted by  $Opt-deliver(m)$  and the traditional total order delivery denoted by  $Adeliver(m)$ .  $Opt-deliver(m)$  occurs upon reception of  $m$ . Even though the order of  $m$  is not yet decided, the processing of the request contained in  $m$  is optimistically started. As in [PS98], the optimism is related to the spontaneous total order property of LANs. If  $Adeliver(m)$  invalidates on some server  $s_i$  the temporary order defined by  $Opt-deliver(m)$ , then the replica  $s_i$  must rollback and undo the processing of request  $m$ . Notice that in this case the inconsistency is *internal* to the server  $s_i$ : no response is sent to a client before the  $Adeliver$  of  $m$ .

This last example leads us to suggest another classification of optimism in the context of active replication and/or Atomic Broadcast:

- a) Optimism that never leads to inconsistency.
- b) Optimism that leads to internal inconsistencies only (server inconsistencies only).
- c) Optimism that leads to external inconsistencies (server and client inconsistencies).

The optimistic Atomic Broadcast algorithm of [PS98] is an example of (a). The optimistic delivery and processing of [KPAS99] is an example of (b). The optimistic concurrency control in the context of transactions is another example of (b) (optimistic concurrency control never leads a

client to see an inconsistent state of the data). The Isis Atomic Broadcast algorithm based on a sequencer [BSS91, KT91] is an example of (c): in some runs the total order delivery of messages can be violated leading clients, in the context of active replication, to receive inconsistent responses (see Section 2.4). The violation of total order can occur with a variable probability, depending on the network/processor load, and on the timeout value chosen for suspecting crashed processes. Despite the potential inconsistencies the algorithm was chosen in Isis for its low cost in absence of failures.

The optimistic active replication technique given in the paper builds on this last algorithm. It prevent however external inconsistencies, even though internal inconsistencies are possible. For this reason the technique is to be classified under category (b). In the next section we briefly recall the Isis Atomic Broadcast algorithm [BSS91].

## 2.4 The Isis sequencer-based atomic broadcast algorithm

Consider a group  $G$  of replicated servers, and a client process issuing an Atomic Broadcast of  $m$  to  $G$ . The sequencer algorithm works as follows (Figure 1(a)):

1. The message  $m$  is sent to the replicas in  $G$ .
2. One of the replicas in  $G$ , called the sequencer, assigns sequence numbers to messages and sends these numbers to  $G$ .
3. Each replica in  $G$  delivers the messages according to their sequence numbers.

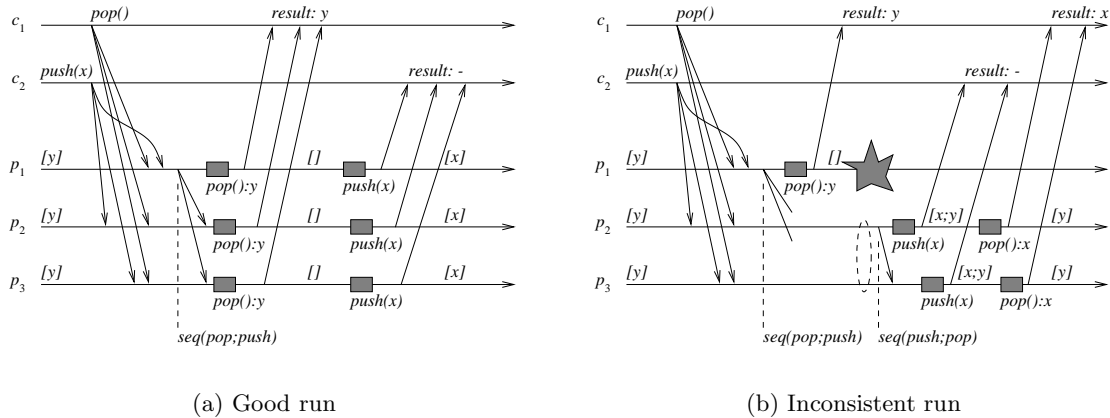


Figure 1: Sequencer-based Atomic Broadcast algorithm

Figure 1(b) gives an example of external inconsistency that can occur. The sequencer crashes or is inconsistently suspected after sending the reply to the client, and the sequencing message is not received by the other replicas (or is received after the sequencer is being suspected). The new sequencer that is elected decides upon a different ordering of messages, making the reply previously received by the client inconsistent.

## 3 System model

For our optimistic active replication algorithm we consider an asynchronous system with processes that communicate by message passing. Processes are either *client* processes or *server* processes.

For simplification, we consider one single replicated service, with server processes denoted by  $\Pi = \{p_1, \dots, p_n\}$ . The client processes, which are not part of  $\Pi$ , are denoted by  $c_1, \dots, c_n$ .

Processes only fail by crashing (i.e., we do not consider Byzantine failures). Processes are connected by reliable channels, defined in terms of the two primitives *send* and *receive*. We also assume that channels are *FIFO*. Moreover, we assume the existence of a *R-multicast*( $m, \Pi$ ) primitive, defined by the following properties: (*Validity*) if a correct process executes *R-multicast*( $m, \Pi$ ), then every correct process in  $\Pi$  eventually R-delivers  $m$ , (*Agreement*) if a correct process R-delivers  $m$ , then all correct processes in  $\Pi$  eventually R-deliver  $m$ , and (*Integrity*) for any message  $m$ , every process R-delivers  $m$  only once, and only if  $m$  was previously R-multicast.

Our optimistic active replication algorithm relies on a consensus *oracle*. It is well known that such an oracle is not implementable in an asynchronous system [FLP85]. However, as shown in [CT96], the consensus oracle is implementable in an asynchronous system augmented with the failure detector  $\diamond\mathcal{S}$  and a majority of correct processes.

## 4 Overview of the algorithm

Like most optimistic algorithms, the optimistic active replication (OAR) algorithm is based on the assumption that failures are infrequent, i.e., the algorithm is optimized for failure-free runs. It uses a lightweight sequencer protocol similar to the protocol described in Section 2.4, which requires a minimal number of communication phases in absence of failures. The originality of the algorithm is that, despite the fact that the replies sent to a client may be different, it guarantees that the client will never “adopt” an inconsistent reply (unlike the algorithm of Section 2.4). The algorithm also includes mechanisms for resolving the temporary inconsistencies that may affect some servers.

To send its request to the servers, a client uses a *Reliable Multicast* primitive, which ensures that if one correct server receives the request, all correct servers eventually receive the request. The client then waits for replies from the servers. Contrary to the usual active replication techniques, the replies might here not be identical. To allow the client to select a “correct” reply, each server reply  $r$  contains an additional *weight* field. This field is the set of servers that endorse reply  $r$ . The client waits for a quorum of replies, and selects the reply according to a majority rule. The rule ensures the selection of the “correct” reply. The details are given later.

The algorithm responsible for ordering the messages among the servers proceeds in a sequence of *epochs*. Each epoch has two *phases*. In phase 1 — the optimistic phase — the algorithm uses a sequencer to optimistically order the messages fast, assuming no failure. The optimistic message delivery is called *Opt-deliver*. As soon as a request message is Opt-delivered by some server  $p$ , it processes the request and generates a reply, which is sent back to the client.

If the sequencer crashes or is wrongly suspected, the algorithm proceeds to phase 2 — the conservative phase — where it uses a different paradigm (based on consensus) to conservatively order messages. The conservative message delivery is called *A-deliver*. As with optimistic delivery, upon A-delivery of a request message the request is immediately processed and the reply sent back to the client. If the sequences of messages Opt-delivered by each server during phase 1 are not identical, the conservative ordering of phase 2 might invalidate the optimistic ordering of phase 1. However, the following safety property holds:

**Majority guarantee.** If a majority of processes Opt-deliver  $m_1$  before  $m_2$ , then no process A-delivers  $m_2$  before  $m_1$ .

For the (rare) cases where the conservative order is different from the optimistic order, we introduce the *Opt-undeliver*( $m$ ) primitive. This primitive notifies the server that message  $m$  has been Opt-delivered in a wrong order, and that the effects induced by the processing of  $m$  must be undone. A

message Opt-undelivered by a correct process will eventually be delivered again (Opt-delivered or A-delivered). The A-delivery of a message can never be undone.

Phase 2 is handled by the problem that we call *Conservative-order* (or simply *Cnsv-order*), which is solved by reduction to a consensus problem. *Cnsv-order* has two input parameters ( $O\_delivered$ ,  $O\_notdelivered$ ), and outputs two sequences of messages ( $Bad$ ,  $New$ ):

$$\{Bad; New\} \leftarrow Cnsv\_order(O\_delivered, O\_notdelivered)$$

For each server  $p$ ,  $O\_delivered$  is the sequence of messages Opt-delivered by  $p$  during the current epoch;  $O\_notdelivered$  is the sequence of messages received but not yet delivered by  $p$ ;  $Bad$  is the sequence that  $p$  has to Opt-undeliver, and  $New$  is the sequence that  $p$  has to A-deliver.

Note that a minority of processes can deliver messages out of order only if the minority is suspected by the majority (e.g., a minority partition cannot communicate with the majority partition). However, this minority partition does not need to be declared faulty and commit suicide, like in the primary partition paradigm [BSS91, VKCD99].

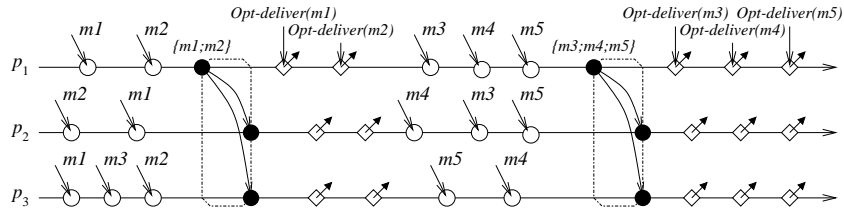


Figure 2: The OAR algorithm with no failure nor suspicion.

Figures 2, 3, and 4 illustrate different runs of the OAR algorithm. Servers  $p_i$  receive incoming message from clients unordered (white circles in the figures). Process  $p_1$  is the sequencer. Black circles represent the reception of the ordering message from the sequencer. Message delivery (Opt-delivery, A-delivery) is represented by white diamonds in the figures, and message undelivery is represented by grey diamonds. Each time a message is delivered to a server, it is immediately processed, and a reply is sent back to the client (this is represented in the figures by the outgoing arrow on the delivery events). In Figure 2, no failure occurs, and processes only execute phase 1 of the OAR algorithm.

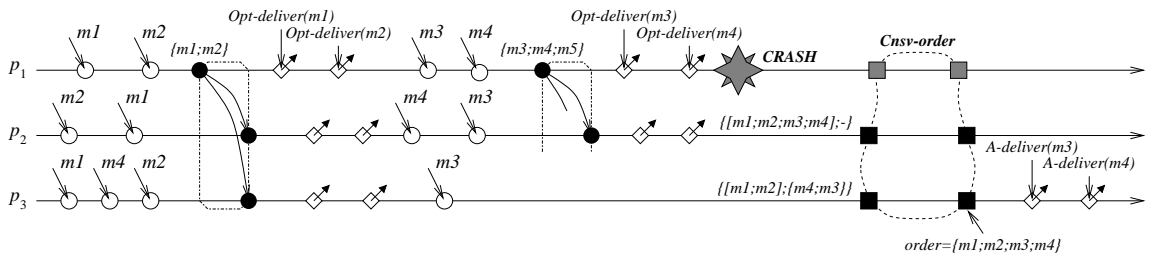


Figure 3: The OAR algorithm with the crash of the sequencer, but no Opt-undelivery.

In Figure 3, the sequencer  $p_1$  fails just after Opt-delivery of  $m_3$  and  $m_4$ . Only processes  $p_2$  receive ordering information from  $p_1$  and Opt-delivers  $m_3$  and  $m_4$ . As a result of the failure, processes proceed to phase 2. In this example, since a majority of processes ( $p_1$  and  $p_2$ ) have Opt-delivered  $m_3$  before  $m_4$ , no process can deliver these messages in a different order. Thus, *Cnsv-order* returns  $Bad = \epsilon$  (the empty sequence),  $New = \epsilon$  for  $p_2$  and  $Bad = \epsilon$ ,  $New = \{m_3; m_4\}$  for  $p_3$ .

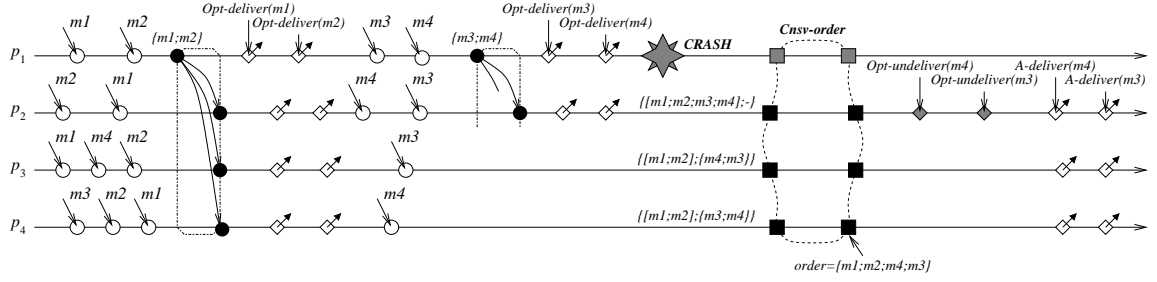


Figure 4: The OAR algorithm with the crash of the sequencer and with Opt-undelivery.

The scenario of Figure 4 is very similar to the one of Figure 3, except that we have four servers. Here since no majority of processes have Opt-delivered the sequence  $m_3$  before  $m_4$ , *Cnsv-order* may decide on a different ordering and return  $Bad = \{m_3; m_4\}$ ,  $New = \{m_4; m_3\}$  at  $p_2$ , and  $Bad = \epsilon$ ,  $New = \{m_4; m_3\}$  at  $p_3$  and  $p_4$ .

## 5 The optimistic active replication algorithm

This section formally describes the OAR algorithm. We first introduce the notation used for expressing the algorithm. Then, we present the client-side and server-side code of the OAR algorithm. We give a formal specification of the *Cnsv-order* primitive, and present its implementation. Finally, the proof of the OAR algorithm is briefly sketched.

### 5.1 Notation

The OAR algorithm manages sequences of messages. This leads us to introduce the following notation. Sequences (of messages) are denoted as follows:  $\{m_1; m_2; m_3\}$ . Sets are denoted as usual using commas:  $\{m_1, m_2, m_3\}$ . An empty sequence is represented by  $\epsilon$ , and an empty set by the usual  $\emptyset$  symbol.

As in [PS98], we use the operators  $\oplus$  and  $\ominus$  for representing concatenation and decomposition of sequences, and the function  $\odot$  for representing the common prefix of a set of sequences. More precisely,  $seq_1 \oplus seq_2$  is the sequence of all the messages from  $seq_1$  followed by all the messages of  $seq_2$ ;  $seq_1 \ominus seq_2$  is the sequence of all messages from  $seq_1$  that are not in  $seq_2$ ; and  $\odot(seq_1, \dots, seq_n)$  is the longest sequence that is a common prefix to  $seq_1, \dots, seq_n$ .

Additionally, we use the symbol  $\uplus$  to designate a function that takes a list of sequences  $seq_1, \dots, seq_n$  as argument and produces a new sequence by appending all sequences together and removing duplicates. More formally,  $\uplus(seq_1, \dots, seq_n)$  is defined recursively as follows:

$$\begin{aligned} \uplus(seq_1) &= seq_1 \\ \uplus(seq_1, \dots, seq_{i+1}) &= \uplus(seq_1, \dots, seq_i) \oplus (seq_{i+1} \ominus \uplus(seq_1, \dots, seq_i)) \end{aligned}$$

We also assume an implicit conversion from a sequence to a set, whenever we use the following set operators:  $\cap$ ,  $\cup$ ,  $\in$ , and  $\notin$ . For instance,  $seq_1 \cap seq_2 = \emptyset$  means that there is no common element in  $seq_1$  and  $seq_2$ .

## 5.2 The client-side algorithm

Figure 5 describes the client-side code of the OAR algorithm. The client R-multicasts its request to all processes of  $\Pi$  (line 2) and waits for a quorum of replies (line 3). The *weight*  $W_i^m$  ( $m$  relates to the request  $m$ ) of  $reply_i^m$  is a set that identifies the servers that endorse  $reply_i^m$ . Said differently, let  $q_i$  be the server that has sent  $reply_i^m$ : the weight  $W_i^m$  contains the identifiers of all servers that  $q_i$  knows to deliver the request in the same order as itself, i.e., to generate the same reply. The value  $k$  identifies the epoch number during which the servers generate the reply.

The client waits until it receives a set of replies that contribute to a total weight greater than or equal to  $\left\lceil \frac{(|\Pi|+1)}{2} \right\rceil$  (majority weight). Our algorithm guarantees that individual replies of an epoch  $k$  that have a minority weight are all identical. Similarly, individual replies that have a majority weight are all identical. If replies with a majority weight are different from the replies that have a minority weight for an epoch  $k$ , the latter cannot have together a total weight greater than or equal to  $\left\lceil \frac{(|\Pi|+1)}{2} \right\rceil$  and the former are considered correct. Therefore, when the set of replies received by the client reaches a majority weight, the client adopts any reply with the largest weight (lines 5–6).

---

```

1: procedure OAR-multicast ( $m, \Pi$ )
2:   R-multicast ( $m, \Pi$ )
3:   wait until for some  $k$ , for  $j$  processes  $q_1, \dots, q_j$  : received  $(reply_i^m, W_i^m, k)$  from  $q_i$  and  $\cup_{i=1}^j W_i^m \geq \left\lceil \frac{(|\Pi|+1)}{2} \right\rceil$ 
4:    $W_{max} \leftarrow$  largest  $W_i^m$  such that received  $(reply_i^m, W_i^m, k)$  from  $q_i$ 
5:    $reply \leftarrow$  select one  $reply_i^m$  such that received  $(reply_i^m, W_{max}, k)$  from  $q_i$ 
6:   return  $reply$ 

```

---

Figure 5: The OAR algorithm: client code

## 5.3 The server-side algorithm

Figure 6 gives the server-side code of the OAR algorithm. Each process of  $\Pi$  maintains a list (i) of incoming messages (*R\_delivered*, line 2), (ii) of messages optimistically delivered during the current epoch (*O\_delivered*, line 3), and (iii) of messages conservatively delivered (*A\_delivered*, line 4). The algorithm progresses through a sequence of epochs, represented by a monotonously increasing integer  $k$  (line 5). An epoch is composed of two parts: the optimistic phase (phase 1) and the conservative phase (phase 2).

The OAR algorithm consists of several tasks. These tasks can execute in any order, but in mutual exclusion. Task 0 receives incoming messages and adds them to *R\_delivered* (line 7). This task is active in phase 1 and in phase 2.

Task 1a, 1b, and 1c are active during phase 1. In Task 1a, the sequencer  $s$  periodically checks whether there are some messages which have been received but not yet ordered (line 8). If so,  $s$  orders these messages in a sequence and sends the sequence to all processes of  $\Pi$  (line 10). In order to simplify the algorithm, we assume that the sequencer immediately delivers this message, and consequently immediately executes Task 1b.

In Task 1b, when some process  $p$  — including the sequencer itself — delivers the sequence  $msgSet^k$  from the sequencer (line 11), it iterates through this sequence (line 16) and performs the following operations: for each message  $m$ , process  $p$  Opt-delivers  $m$ , i.e., processes the request and generates the reply (line 17), adds  $m$  to *O\_delivered* (line 18), and sends a reply to the client (line 19). The weight (lines 12–15) is equal to  $\{s\}$  if  $p$  is the sequencer, and to  $\{s, p\}$  otherwise ( $p$  knows that  $s$  delivers  $m$  in the same order).



---

```

1: Initialization:
2:    $R\_delivered \leftarrow \epsilon$ 
3:    $A\_delivered \leftarrow \epsilon$ 
4:    $O\_delivered \leftarrow \epsilon$ 
5:    $k \leftarrow 0$ 

6: when  $R\_deliver(m)$  {Task 0: buffer incoming client message}
7:    $R\_delivered \leftarrow R\_delivered \oplus \{m\}$ 

8: when  $p = s$  and  $(R\_delivered \ominus A\_delivered) \ominus O\_delivered \neq \epsilon$  {Task 1a: sequencer orders messages}
9:    $O\_notdelivered \leftarrow (R\_delivered \ominus A\_delivered) \ominus O\_delivered$ 
10:  send  $(k, O\_notdelivered)$  to all

11: when  $deliver(k, msgSet^k)$  {Task 1b: processes opt-deliver messages}
12:  if  $p = s$  then
13:     $W \leftarrow \{s\}$ 
14:  else
15:     $W \leftarrow \{p, s\}$ 
16:  for all  $m \in msgSet^k$  do
17:     $reply \leftarrow Opt\_deliver(m)$ 
18:     $O\_delivered \leftarrow O\_delivered \oplus \{m\}$ 
19:    send  $(reply, W, k)$  to  $sender(m)$ 

20: when  $s \in \mathcal{D}_p$  {Task 1c: suspicion}
21:   $R\_broadcast(k, PHASEII)$  to all

22: when  $R\_deliver(k, PHASEII)$  {Task 2: conservative ordering}
23:   $O\_notdelivered \leftarrow (R\_delivered \ominus A\_delivered) \ominus O\_delivered$ 
24:   $\{Bad, New\} \leftarrow Cnsv\_order(k, O\_delivered, O\_notdelivered)$ 
25:  for all  $m \in Bad$  do
26:     $Opt\_undeliver(m)$ 
27:  for all  $m \in New$  do
28:     $reply \leftarrow A\_deliver(m)$ 
29:    send  $(reply, \Pi, k)$  to  $sender(m)$ 
30:   $A\_delivered \leftarrow A\_delivered \oplus (O\_delivered \ominus Bad) \oplus New$ 
31:   $O\_delivered \leftarrow \epsilon$ 
32:   $k \leftarrow k + 1$ 

```

---

Figure 6: The OAR algorithm: code of server process  $p$

In Task 1c, when a process suspects the sequencer to have failed, it R-broadcasts a PHASEII message to notify the other processes to proceed to the conservative phase 2 (line 21).

Task 2 is the task of the conservative phase 2. Process  $p$  proceeds to that phase upon delivery of a PHASEII message (line 22). Process  $p$  then invokes the  $Cnsv\_order$  function, which conservatively orders messages (line 23). The function  $Cnsv\_order$  takes as argument two sequences: the sequence of messages Opt-delivered by  $p$  during epoch  $k$  ( $O\_delivered$ ), and the sequence of messages that have been R-delivered by  $p$  but not yet ordered. The function returns two values: a sequence of messages that  $p$  Opt-delivered in the wrong order ( $Bad$ ) and a sequence of messages that have just been conservatively ordered but not yet delivered ( $New$ ). The  $Cnsv\_order$  function is defined in Section 5.4.

In the (unlikely) event of the sequence  $Bad$  being not empty, process  $p$  first Opt-undelivers these messages<sup>2</sup> (line 26). Then  $p$  A-delivers sequentially all the messages in  $New$  (line 28) and sends a reply with a weight equals to  $\Pi$  to the client (line 29). The weight indicates agreement of the order that has been decided. Finally,  $p$  adds the messages delivered during epoch  $k$  to  $A\_delivered$ , clears

---

<sup>2</sup>Although not shown in the algorithm, undelivery of messages should generally be performed in the reverse order of delivery, i.e., starting by the last message of  $Bad$ .

$O\_delivered$ , and proceeds to epoch  $k + 1$  (lines 30–32).

In the code of the algorithm, we assume that the sequencer  $s$  does not change. This may be a problem if  $s$  fails and does not recover immediately. A simple solution to that problem consists in using a rotating coordinator. The sequencer is initially set to the first process of  $\Pi$ , i.e.,  $s \leftarrow 0$  (for simplicity, we designate processes by their position in  $\Pi$ ). At the end of phase 2, each process defines the new sequencer to be  $s \leftarrow (s + 1) \bmod |\Pi|$ . This scheme prevent a crashed sequencer to continuously slow down the system.

**Remark.** In the OAR algorithm of Figure 6, execution of phase 2 allows to “forget” about messages Opt-delivered (line 31). So, if phase 2 is executed only rarely, the sequence  $O\_delivered$  can become extremely long, which might slow down the execution of the next instance of *Cnsv-order* in phase 2. This problem can easily be solved, e.g., by having the sequencer R-broadcast a PHASEII message on a regular basis (e.g., every  $n$  requests or every  $t$  seconds) to explicitly execute phase 2. More lightweight solutions to garbage collect the  $O\_delivered$  sequence also exist, but are not detailed here.

#### 5.4 Specification of *Cnsv-order*

We specify the *Cnsv-order* problem by the following properties, which are commented below:

**Termination.** If a correct process  $p$  calls *Cnsv-order* then eventually  $p$  gets the result  $\{Bad_p; New_p\}$ .

**Agreement.** For any two correct processes  $p$  and  $q$ ,  $(O\_delivered_p \oplus Bad_p) \oplus New_p = (O\_delivered_q \oplus Bad_q) \oplus New_q$ .

**Unicity.** For all processes  $p$ , we have  $New_p \cap (O\_delivered_p \oplus Bad_p) = \emptyset$ .

**Non-triviality.** If for a majority of processes  $q$ ,  $m \in O\_delivered_q \cup O\_notdelivered_q$ , then for all correct processes  $p$ , we have  $m \in (O\_delivered_p \oplus Bad_p) \oplus New_p$ .

**Validity.** If for any process  $p$ ,  $m \in New_p$ , then for at least one process  $q$ , we have  $m \in O\_delivered_q \cup O\_notdelivered_q$ .

**Undo legality.** For all processes  $p$ , we have  $(O\_delivered_p \oplus Bad_p) \oplus Bad_p = O\_delivered_p$ .

**Undo consistency.** For all processes  $p$ , if  $m \in Bad_p$  then for a majority of processes  $q$ , we have  $m \notin O\_delivered_q$ .

The termination property ensures progress. The agreement property ensures agreement on the sequence of messages delivered during epoch  $k$ . The unicity property forbids a message to be delivered twice during epoch  $k$  (a message can be Opt-delivered and A-delivered only if it is meanwhile Opt-undelivered, i.e., in the sequence  $Bad$ ). The non-triviality property states that any message that has been received by a majority of processes during epoch  $k$  will also be delivered during epoch  $k$ , and thus prohibits the trivial solution where  $New_p$  is always empty. The validity property prevents arbitrary messages from being in  $New_p$ . The undo legality guarantees that  $Bad_p$  is well formed, i.e., a suffix of  $O\_delivered_p$ . The undo consistency property prevents messages from being in  $Bad_p$  if they have been Opt-delivered by a majority of processes.

The above properties are sufficient to ensure the correctness of the OAR algorithm. However, we add the following property, which guarantees that no optimistic delivery will be unnecessarily undone.

**Undo thriftiness.** For all processes  $p$ , we have  $\odot(Bad_p, New_p) = \epsilon$ .

## 5.5 Implementation of *Cnsv-order*

The *Cnsv-order* problem can be solved by reduction to a consensus problem. Recall that consensus is defined in terms of two primitives  $propose(v)$  and  $decide(v)$ , and specified as follows:

**Termination.** Each correct process eventually decides.

**Validity.** If a process executes  $decide(v)$ , then some process has executed  $propose(v)$ .

**Agreement.** No two correct processes decide differently.

For solving the *Cnsv-order* problem, we use a slightly different specification. The *Validity* property is replaced by the following *Maj-validity* property, in which the decision  $V$  is a sequence of initial values:

**Maj-validity.** If a process executes  $decide(V)$ , then  $V$  is a sequence of values such that, for a majority of processes  $p_i$ , if  $p_i$  has executed  $propose(v_i)$ , then  $v_i \in V$ .

Said differently, the decision sequence contains the initial value of a majority of processes. The consensus with the Maj-validity property can be solved by minor modifications to the consensus algorithm based on  $\diamond\mathcal{S}$  [CT96]. For a description of these modifications, see [Fel98].

---

```

1: procedure Cnsv-order ( $k, O\_dlv, O\_notdlv$ )
2:    $Bad \leftarrow New \leftarrow \epsilon$ 
3:    $propose(k, \{O\_dlv; O\_notdlv\})$ 
4:   wait until  $decide(k, D^k)$  {  $D^k$  is a sequence  $\{(dlv_1, notdlv_1); (dlv_2, notdlv_2); \dots\}$  }
5:    $dlv_{max} \leftarrow$  longest  $dlv_i$  such that  $\{dlv_i; notdlv_i\} \in D^k$ 
6:   if  $O\_dlv = \odot(O\_dlv, dlv_{max})$  then
7:      $New \leftarrow dlv_{max} \odot O\_dlv$ 
8:      $Good \leftarrow O\_dlv$ 
9:   else
10:     $Good \leftarrow \odot(O\_dlv, dlv_{max})$ 
11:     $Bad \leftarrow O\_dlv \oplus Good$ 
12:     $notdlv \leftarrow \oplus_i(notdlv_i \text{ such that } \{dlv_i; notdlv_i\} \in D^k)$ 
13:     $notdlv \leftarrow notdlv \oplus dlv_{max}$ 
14:     $New \leftarrow New \oplus notdlv$ 
15:    if  $\odot(Bad, New) \neq \epsilon$  then
16:       $prefix = \odot(Bad, New)$ 
17:       $Good \leftarrow Good \oplus prefix$ 
18:       $Bad \leftarrow Bad \oplus prefix$ 
19:       $New \leftarrow New \oplus prefix$ 
20:    return  $\{Bad; New\}$ 

```

---

Figure 7: The conservative ordering procedure

The implementation of the *Cnsv-order* function is given in Figure 7. Each process computes three sequences: *Good* (messages Opt-delivered in the right order during epoch  $k$ ), *Bad* (messages Opt-delivered in the wrong order during epoch  $k$ ), and *New* (messages to A-deliver during epoch  $k$ ). At line 3, the processes propose their initial value for consensus, which consists of a pair of sequences of messages  $(O\_dlv, O\_notdlv)$ . The decision at line 4 is a sequence of pairs denoted by  $D^k \equiv \{(dlv_1, notdlv_1); (dlv_2, notdlv_2); \dots\}$ . Upon decision,  $p$  selects the longest sequence  $dlv_i$  denoted by  $dlv_{max}$  (line 5).<sup>3</sup>

---

<sup>3</sup>The sequences  $dlv_i$  can differ only by their length, i.e., given any two sequences  $dlv_i, dlv_j$ , if they are not equal, one is a prefix of the other.

If for  $p$ ,  $O\_dlv$  is a subsequence of  $dlv_{max}$  (line 6), then  $p$  sets  $New$  equal to the subsequence of  $dlv_{max}$  that it did not yet deliver (line 7).  $Good$  is set to the sequence already delivered (line 8).

However, if for  $p$ ,  $dlv_{max}$  is shorter than  $O\_dlv$  (line 6) (i.e.,  $p$ 's initial value is not contained in the decision), there may be a risk of inconsistent ordering. In this case,  $Good$  is set to the sequence of messages delivered in the correct order (line 10), and  $Bad$  is set to the sequence of wrongly-ordered messages (line 11).

Process  $p$  then generates deterministically a sequence  $notdlv$  with all  $notdlv_i$  sequences in the decision  $D^k$  of the consensus  $\#k$  (line 12), makes sure that this sequence does not contain any message correctly Opt-delivered or already scheduled for delivery (line 13), and adds this sequence to  $New$  (line 14).

Finally, at line 15,  $p$  checks if there if  $Bad$  and  $New$  have a common prefix, i.e., if it will undeliver some messages and re-deliver them in the same order. This may happen if some messages are added to  $Bad$  because they are not part of any  $dlv_i$  in  $D^k$ , but are incidentally rescheduled for delivery in the same order. In that case,  $p$  adds these messages to  $Good$  and removes them from both  $New$  and  $Bad$  (lines 17–19). This ensures that the implementation of *Cnsv-order* satisfies the Undo thriftiness property of Section 5.4.

## 5.6 Proof of the algorithm

The proof of the OAR algorithm is given in the Appendix. The proof of *Cnsv-order* consists in proving the properties of Section 5.4. The rest of the proof of the OAR algorithm consists in proving the following results. In the proofs, we assume that the *reply* sent by the active replicated servers to the client is a *number*, whose value indicates the order of processing of the client request.

**Proposition 1 (Validity of request handling).** *If a server  $p$  executes either (1) “Opt-deliver( $m$ )” (Fig. 6, line 17) or (2) “A-deliver( $m$ )” (Fig. 6, line 28), then a client  $c$  has previously executed “OAR-multicast( $m, \Pi$ )” (Fig. 5).*

**Proposition 2 (At most once request handling – 1).** *During epoch  $k$ , if a server  $p$  executes both “Opt-deliver( $m$ )” (Fig. 6, line 17) and “A-deliver( $m$ )” (Fig. 6, line 28), then it also executes “Opt-undeliver( $m$ )” (Fig. 6, line 26) during epoch  $k$ .*

**Proposition 3 (At most once request handling – 2).** *During epoch  $k$ , if a server  $p$  either (1) executes “Opt-deliver( $m$ )” (Fig. 6, line 17) but not “Opt-undeliver( $m$ )” (Fig. 6, line 26), or (2) executes “A-deliver( $m$ )” (Fig. 6, line 28), then  $p$  will execute neither “Opt-deliver( $m$ )”, nor “A-deliver( $m$ )” in any epoch  $k' > k$ .*

**Proposition 4 (At least once request handling).** *If a correct client executes “OAR-multicast( $m, \Pi$ )” or if a correct server receives request  $m$  (Fig. 6, line 6), then there exists an epoch  $k$  such that in epoch  $k$  each correct server  $p$  eventually either (1) executes “Opt-deliver( $m$ )” (Fig. 6, line 17) but not “Opt-undeliver( $m$ )” (Fig. 6, line 26), or (2) executes “A-deliver( $m$ )” (Fig. 6, line 28).*

**Proposition 5 (Total order).** *Consider two correct servers  $p, q$  and message  $m$ :*  
(1) *If both servers execute “reply  $\leftarrow$  A-deliver( $m$ )” (Fig. 6, line 28), then the two “reply” values at line 17 (Fig. 6) are equal.*  
(2) *If both servers execute “reply  $\leftarrow$  Opt-deliver( $m$ )” (Fig. 6, line 17) without executing “Opt-undeliver( $m$ )” (Fig. 6, line 26) during the same epoch, then the two reply values are equal.*  
(3) *If one server executes “reply  $\leftarrow$  Opt-deliver( $m$ )” (Fig. 6, line 17) without executing “Opt-undeliver( $m$ )” (Fig. 6, line 26) during the same epoch, and the other server executes “reply  $\leftarrow$  A-deliver( $m$ )” (Fig. 6, line 28), then the two “reply” values are equal.*

Although not necessary for the correctness of the OAR algorithm, the following proposition guarantees that processes do not Opt-undeliver messages unless necessary.

**Proposition 6 (No unnecessary Opt-undeliver).** *For each correct server  $p$ , if  $p$  both executes “ $r_{opt} = \text{reply} \leftarrow \text{Opt-deliver}(m)$ ” (Fig. 6, line 17) and “ $r_a = \text{reply} \leftarrow \text{A-deliver}(m)$ ” (Fig. 6, line 28) during epoch  $k$  with “ $r_{opt} = r_a$ ”, then there exists a message  $m'$  such that  $p$  executes both “ $r'_{opt} = \text{reply} \leftarrow \text{Opt-deliver}(m')$ ” and “ $r'_a = \text{reply} \leftarrow \text{A-deliver}(m')$ ” during epoch  $k$  with “ $r'_{opt} < r_{opt}$ ” and “ $r'_{opt} \neq r'_a$ ”.*

The following proposition guarantees that clients always adopt consistent replies.

**Proposition 7 (External consistency).** *If a client  $c$  executes “OAR-multicast( $m, \Pi$ )” (Fig. 5) and gets “ $\text{reply} = r$ ” (Fig. 5, line 5), then no correct processes ever*  
(1) *executes “ $\text{reply} \leftarrow \text{A-deliver}(m)$ ” (Fig. 6, line 28) with “ $\text{reply} \neq r$ ”, or*  
(2) *executes “ $\text{reply} \leftarrow \text{Opt-deliver}(m)$ ” (Fig. 6, line 26) with “ $\text{reply} \neq r$ ” without executing “Opt-undeliver( $m$ )” (Fig. 6, line 26) during the same epoch.*

## 6 Conclusion

Our optimistic active replication algorithm solves Atomic Broadcast as a subproblem.<sup>4</sup> However, the originality of the OAR algorithm is to handle Atomic Broadcast as a white box, rather than as black box as usually done. This allows us to have an algorithm that is both efficient in terms of latency, while preserving consistency at the client level. Similarly to sequencer-based Atomic Broadcast algorithms (e.g., [BSS91, KT91]), our algorithm requires only one phase for ordering messages in absence of failures, but unlike sequencer-based protocols it prevents inconsistencies that may occur with these algorithms.

Reconciliation among the servers is handled thanks to the *Opt-undeliver* primitive. The probability of having to Opt-undeliver a message is very low. It requires a combination of three events: (1) the sequencer  $s$  fails or is suspected in such a way that only a minority of processes (call them  $P_{min}$ ) have received ordering information from  $s$ , (2) no process of  $P_{min}$  has its initial value in the decision of the consensus,<sup>5</sup> and (3) the messages Opt-delivered only by the processes of  $P_{min}$  are conservatively ordered differently by *Cnsv-order*. Events (1) and (2) can happen for example if  $s \in P_{min}$  and  $P_{min}$  is partitioned from other processes of  $\Pi$ .

The OAR algorithm is particularly well adapted to transactional environment, where the effect of some operations can be undone (rollback). Before Opt-delivering a message, each server  $p$  can start a new transaction or declare a save-point. During phase 2,  $p$  can commit the transactions associated with messages in *Good<sub>p</sub>* (see the *Cnsv-order* algorithm), and abort those associated with messages in *Bad<sub>p</sub>*. Messages A-delivered by  $p$  never need to be undelivered and the associated transactions can be committed immediately.

We believe that the OAR algorithm presented in this paper offers a good compromise between efficiency (low latency) and consistency, by not trying to preserve server consistency by all means, but always ensuring consistency at the client level.

<sup>4</sup>Where the atomic delivery of a message corresponds to the message being either (1) A-delivered or (2) Opt-delivered but not Opt-undelivered.

<sup>5</sup>When using an implementation of the consensus protocol that collects estimates from all non-suspected processes, this means that all processes of  $P_{min}$  are suspected.

## References

- [BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Distributed Database Systems*. Addison-Wesley, 1987.
- [BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic multicast. *Transactions on Computer Systems*, 9(3):272–314, August 1991.
- [CT96] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.
- [Déf00] Xavier Défago. *Agreement-Related Problems: From Semi-Passive Replication to Totally Ordered Broadcast*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, August 2000. Number 2229.
- [DSS98] X. Défago, A. Schiper, and N. Sergent. Semi-passive Replication. In *17th IEEE Symp. on Reliable Distributed Systems (SRDS-17)*, pages 43–58, West Lafayette, USA, October 1998.
- [Fel98] P. Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998. Number 1867.
- [FLP85] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):217–246, 1985.
- [Gif79] D.K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, pages 150–162, 1979.
- [GS97] R. Guerraoui and A. Schiper. Software-Based Replication for Fault Tolerance. *IEEE Computer*, 30(4):68–74, April 1997.
- [HT93] V. Hadzilacos and S. Toueg. *Distributed Systems*, chapter 5: Fault-Tolerant Broadcasts and Related Problems, pages 97–145. Addison-Wesley, 2nd edition, 1993.
- [KPAS99] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over Optimistic Atomic Broadcast Protocols. In *IEEE 19th Intl. Conf. Distributed Computing Systems (ICDCS-19)*, pages 424–431, June 1999.
- [KT91] M. F. Kaashoek and A. S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS-11)*, pages 222–230, Arlington, TX, USA, May 1991.
- [Ped99] F. Pedone. *The Database State Machine and Group Communication Issues*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, December 1999. Number 2090.
- [PS98] F. Pedone and A. Schiper. Optimistic Atomic Broadcast. In *12th. Intl. Symposium on Distributed Computing (DISC’98)*, pages 318–332. Springer Verlag, LNCS 1499, September 1998.
- [RG93] T.W. Page Jr. R.G. Guy, G.J. Popek. Consistency algorithms for optimistic replication. In *1st IEEE Int. Conference on Network Protocols*, October 1993.

- [VKCD99] R. Vitenberg, I. Keidar, G.V. Chockler, and D. Dolev. Group Communication Specifications: A Comprehensive Study. Tr, Dept. of Computer Science, Technion, Israel, September 99.

## A Proofs

This section presents the proofs of the OAR algorithm, with the assumption of a majority of correct servers. We first prove that the OAR algorithm is correct, assuming that *Cnsv-order* satisfies the properties given in Section 5.4. After that, we show that the *Cnsv-order* algorithm indeed ensures these properties.

To prove the OAR algorithm, we consider that the *reply* sent by the active replicated servers to the client is a *number*, whose value indicates the order of processing of the client request. In other words, if a server  $p$  sends back  $reply = r$ , then the client request was handled by  $p$  at position  $\#r$ .

We generally omit  $\Pi$  in the expression of the proofs (e.g., instead of writing “a server  $p$  of  $\Pi$ ”, we simply write “a server  $p$ ”).

### A.1 Correctness of the OAR algorithm: the server side

In this section we assume that all line numbers refer to Figure 6, unless specified otherwise.

**Lemma 1 (Atomicity of request delivery).** *If (1) a correct client  $c$  executes “OAR-multicast( $m, \Pi$ )” (Fig. 5), or (2) if a correct server receives request  $m$  (line 6), then every correct server eventually receives request  $m$ .*

*Proof.* Client  $c$  executes *R-multicast*( $m, \Pi$ ) (Fig. 5, line 2). The proof is immediate from the validity and agreement properties of *R-multicast*.  $\square$

**Proposition 1 (Validity of request handling).** *If a server  $p$  executes either (1) “Opt-deliver( $m$ )” (line 17) or (2) “A-deliver( $m$ )” (line 28), then a client  $c$  has previously executed “OAR-multicast( $m, \Pi$ )” (Fig. 5).*

*Proof.* In case (1), a message is Opt-delivered (line 17) only if it is part of the sequence  $msgSet^k$  sent by the sequencer  $s$  at line 10 and delivered at line 11. If  $m \in msgSet^k$ , then  $m \in R\_delivered$  (line 9) and  $s$  has previously R-delivered  $m$  (lines 6–7). The integrity property of *R-multicast* guarantees that  $m$  was previously R-multicast (Fig. 5, line 2) and thus that a client  $c$  has previously executed “OAR-multicast( $m, \Pi$ )”.

In case (2), we distinguish case (2a) where  $m$  was previously Opt-delivered by  $p$ , and case (2b) where  $m$  has never been Opt-delivered by  $p$ .

Case (2a) can be trivially reduced to case (1). In case (2b), process  $p$  A-delivers  $m$  at line 28 and thus  $m \in New_p$  at line 24. The validity property of *Cnsv-order* guarantees that for at least one process  $q$ ,  $m \in O\_delivered_q \cup O\_notdelivered_q$ . If  $m \in O\_delivered_q$ , then as in case (1) we conclude that a client  $c$  has previously executed “OAR-multicast( $m, \Pi$ )”. If  $m \in O\_notdelivered_q$ , then  $m \in R\_delivered_q$  (line 23) and  $q$  has previously R-delivered  $m$  (lines 6–7). As above, we conclude that a client  $c$  has previously executed “OAR-multicast( $m, \Pi$ )”.  $\square$

**Proposition 2 (At most once request handling – 1).** *During epoch  $k$ , if a server  $p$  executes both “Opt-deliver( $m$ )” (line 17) and “A-deliver( $m$ )” (line 28), then it also executes “Opt-undeliver( $m$ )” (line 26) during epoch  $k$ .*

*Proof.* Assume by contradiction that some server  $p$  Opt-delivers and A-delivers  $m$  during epoch  $k$ , but does not Opt-undeliver  $m$ . This means that  $m \in O\_delivered_p$  and  $m \in New_p$ , but  $m \notin Bad_p$ . This is in contradiction with the unicity property of *Cnsv-order*.  $\square$

**Proposition 3 (At most once request handling – 2).** *During epoch  $k$ , if a server  $p$  either (1) executes “Opt-deliver( $m$ )” (line 17) but not “Opt-undeliver( $m$ )” (line 26), or*



(2) executes “A-deliver( $m$ )” (line 28),  
then  $p$  will execute neither “Opt-deliver( $m$ )”, nor “A-deliver( $m$ )” in any epoch  $k' > k$ .

*Proof.* Client  $c$  executes  $R$ -multicast( $m, \Pi$ ) (Fig. 5, line 2). The integrity property of  $R$ -multicast guarantees that all servers will deliver  $m$  at most once (line 6).

By contradiction, assume that some server  $p$  (i) Opt-delivers (but does not Opt-undeliver) or A-delivers  $m$  during epoch  $k$ , and (ii) Opt-delivers the same message  $m$  during epoch  $k' > k$ . Because of (ii), there must exist some process  $q$  such that  $m$  is in the sequence  $msgSet^{k'}$  delivered at line 11 (because  $m$  is Opt-delivered at line 17). So  $m$  is in the sequence sent by the sequencer  $s$  of epoch  $k'$  (line 10). By line 9,  $s$  has neither A-delivered, nor Opt-delivered (but not Opt-undelivered)  $m$ . By the agreement property of  $Cnsv$ -order, no process has A-delivered or Opt-delivered (but not Opt-undelivered)  $m$ . A contradiction with (i).

Now assume that some server  $p$  (i) Opt-delivers (but does not Opt-undeliver) or A-delivers  $m$  during epoch  $k$ , and (ii) A-delivers the same message  $m$  during epoch  $k' > k$ . By the validity property of  $Cnsv$ -order, there must exist some process  $q$  such that  $m \in O\_delivered_q \cup O\_notdelivered_q$  during epoch  $k'$ . By line 23,  $q$  has neither delivered, not Opt-delivered (but not Opt-undelivered)  $m$ . By the agreement property of  $Cnsv$ -order, no process has A-delivered or Opt-delivered (but not Opt-undelivered)  $m$ . A contradiction with (i).  $\square$

**Proposition 4 (At least once request handling).** *If a correct client executes “OAR-multicast( $m, \Pi$ )” or if a correct server receives request  $m$  (line 6), then there exists an epoch  $k$  such that in epoch  $k$  each correct server  $p$  eventually either*

- (1) executes “Opt-deliver( $m$ )” (line 17) but not “Opt-undeliver( $m$ )” (line 26), or
- (2) executes “A-deliver( $m$ )” (line 28).

*Proof.* By Lemma 1, all correct servers eventually receive  $m$ . By assumption, there are a majority of correct servers. Let  $k$  be the epoch such that all faulty processes have crashed, and all correct servers have R-delivered  $m$  (line 6). Let  $s$  be the (correct) sequencer of epoch  $k$ .

We consider two cases: (1)  $s$  has already Opt-delivered (but not Opt-undelivered) or A-delivered  $m$  in an epoch  $k' < k$ , or (2)  $s$  has neither Opt-delivered nor A-delivered  $m$  in an epoch  $k' < k$ .

In case (1), epoch  $k'$  was necessarily completed by an execution of phase 2, and by the agreement property of  $Cnsv$ -order, every process that has proceeded to epoch  $k' + 1 \leq k$  has also Opt-delivered (but not Opt-undelivered) or A-delivered  $m$  in epoch  $k'$ .

In case (2), we distinguish case (2a) where epoch  $k$  never terminates, and case (2b) where epoch  $k$  terminates with an execution of phase 2.

In case (2a),  $s$  eventually sends  $(k, O\_notdelivered)$  to all servers with  $m \in O\_notdelivered$  (line 10). Because channels are reliable and no process proceeds to phase 2, every correct server eventually delivers  $(k, O\_notdelivered)$  (line 11) and Opt-delivers  $m$  (line 18). Because no process proceeds to phase 2, no process Opt-undelivers  $m$ .

In case (2b), as every correct server has R-delivered  $m$ , for every correct server we have  $m \in O\_delivered \cup O\_notdelivered$ . By the non-triviality property of  $Cnsv$ -order, for all correct servers  $p$  we have  $m \in (O\_delivered_p \ominus Bad_p) \oplus New_p$ . So  $p$  either has Opt-delivered  $m$  and did not Opt-undeliver  $m$  in epoch  $k$ , or  $p$  A-delivers  $m$  in epoch  $k$ .  $\square$

**Proposition 5 (Total order).** *Consider two correct servers  $p, q$  and message  $m$ :*

- (1) *If both servers execute “reply  $\leftarrow$  A-deliver( $m$ )” (line 28), then the two “reply” values at line 17 are equal.*
- (2) *If both servers execute “reply  $\leftarrow$  Opt-deliver( $m$ )” (line 17) without executing “Opt-undeliver( $m$ )” (line 26) during the same epoch, then the two reply values are equal.*
- (3) *If one server executes “reply  $\leftarrow$  Opt-deliver( $m$ )” (line 17) without executing “Opt-undeliver( $m$ )”*

(line 26) during the same epoch, and the other server executes “ $\text{reply} \leftarrow A\text{-deliver}(m)$ ” (line 28), then the two “reply” values are equal.

*Proof.* Assume that each server  $p$  manages an integer  $\text{reply}_p$ , initially equal to 0, and incremented whenever a message is Opt-delivered or A-delivered, and decremented each time a message is Opt-undelivered.

During each epoch  $k$ , the agreement property of *Cnsv-order* guarantees that each process delivers (i.e., (i) Opt-delivers but does not Opt-undeliver, or (ii) A-delivers) the same number of message:  $|(O\_delivered_p \ominus Bad_p) \oplus New_p|$ . Therefore, at the beginning of each epoch, for any two processes  $p$  and  $q$ , we have  $\text{reply}_p = \text{reply}_q$ .

*Case (1).* By Propositions 3 and 4 we have that  $p$  and  $q$  A-deliver  $m$  during the same epoch  $k$ . Since both processes A-deliver  $m$ , by Proposition 2, we have  $m \notin (O\_delivered \ominus Bad)$  and  $m \in New$  for both  $p$  and  $q$ . Let  $\#m$  be the position of  $m$  in  $(O\_delivered_p \ominus Bad_p) \oplus New_p$ . The agreement property of *Cnsv-order* implies that  $m$  is also at position  $\#m$  in  $(O\_delivered_q \ominus Bad_q) \oplus New_q$ . Since  $\text{reply}_p = \text{reply}_q$  at the beginning of epoch  $k$ , we have at  $p$  and  $q$ :  $\#r = \text{reply}_p + \#m = \text{reply}_q + \#m$ .

The proofs of case (2) and case (3) are based on the same idea as the proof of case (1).

*Case (2).* Propositions 3 and 4 guarantee that  $p$  and  $q$  Opt-deliver (without Opt-undelivery)  $m$  during the same epoch  $k$ . Since  $p$  and  $q$  Opt-deliver but do not Opt-undeliver  $m$ , by Proposition 2, we have  $m \in O\_delivered$ ,  $m \notin Bad$ , and  $m \notin New$  for both  $p$  and  $q$ . Let  $\#m$  be the position of  $m$  in  $O\_delivered_p$ . The agreement property of *Cnsv-order* implies that  $m$  is also at position  $\#m$  in  $O\_delivered_q$ . Since  $\text{reply}_p = \text{reply}_q$  at the beginning of epoch  $k$ , we have  $\#r = \text{reply}_p + \#m = \text{reply}_q + \#m$ .

*Case (3).* Let  $p$  be the process that Opt-delivers (but does not Opt-undeliver)  $m$ , and  $q$  the process that A-delivers  $m$ . Propositions 3 and 4 guarantee that  $p$  Opt-delivers (without Opt-undelivering) during the same epoch  $k$  as  $q$  Opt-delivers  $m$ . Using the same argument as in cases (1) and (2), we have for  $p$ :  $m \in O\_delivered_p$ ,  $m \notin Bad_p$ , and  $m \notin New_p$ , and for  $q$ :  $m \notin (O\_delivered_q \ominus Bad_q)$  and  $m \in New_q$  for  $q$ . Let  $\#m$  be the position of  $m$  in  $O\_delivered_p$ . The agreement property of *Cnsv-order* implies that  $m$  is also at position  $\#m$  in  $(O\_delivered_q \ominus Bad_q) \oplus New_q$ . Since  $\text{reply}_p = \text{reply}_q$  at the beginning of epoch  $k$ , we have  $\#r = \text{reply}_p + \#m = \text{reply}_q + \#m$ .  $\square$

\* \* \*

The Propositions 1 to 5 prove the correctness of the OAR algorithm from the point of view of the servers. Proposition 6 below shows that our algorithm does not lead to the execution of unnecessary Opt-undeliver operations.

**Proposition 6 (No unnecessary Opt-undeliver).** *For each correct server  $p$ , if  $p$  both executes “ $r_{\text{opt}} = \text{reply} \leftarrow \text{Opt-deliver}(m)$ ” (line 17) and “ $r_a = \text{reply} \leftarrow A\text{-deliver}(m)$ ” (line 28) during epoch  $k$  with “ $r_{\text{opt}} = r_a$ ”, then there exists a message  $m'$  such that  $p$  executes both “ $r'_{\text{opt}} = \text{reply} \leftarrow \text{Opt-deliver}(m')$ ” and “ $r'_a = \text{reply} \leftarrow A\text{-deliver}(m')$ ” during epoch  $k$  with “ $r'_{\text{opt}} < r_{\text{opt}}$ ” and “ $r'_{\text{opt}} \neq r'_a$ ”.*

*Proof.* If  $p$  executes both “ $\text{Opt-deliver}(m)$ ” and “ $A\text{-deliver}(m)$ ” during epoch  $k$ , then  $p$  also executes “ $\text{Opt-undeliver}(m)$ ” during epoch  $k$  (Proposition 2). Therefore, we have  $m \in \text{Opt\_delivered}_p$ ,  $m \in Bad_p$ , and  $m \in New_p$ . We have to prove that a previous message  $m'$  of epoch  $k$  was Opt-delivered in the wrong order.

We prove first the following intermediary result: message  $m$  has the same position in  $Bad_p$  and  $New_p$  if and only if  $m$  is Opt-delivered and A-delivered at the same rank. The undo legality property of *Cnsv-order* guarantees that  $Bad_p$  is a suffix of  $O\_delivered_p$ . In other words, the messages are ordered in  $Bad_p$  in the same order as they have been Opt-delivered. The result follows

immediately from the fact that after messages from  $Bad_p$  have been undelivered, messages from  $New_p$  are sequentially A-delivered.

Now assume by contradiction that there is no message  $m'$  such that  $m'$  precedes  $m$  in  $Bad_p$  and  $m'$  is not at the same position in  $Bad_p$  and  $New_p$ . This means that all messages (if any) that precede  $m$  in  $Bad_p$  are at the same position in  $New_p$ , and thus  $\odot(Bad_p, New_p) \neq \epsilon$ . This contradicts the undo thriftiness property of *Cnsv-order*.  $\square$

## A.2 Correctness of the OAR algorithm: the client side

The next proposition guarantees that a client always “sees” a consistent reply to its request.

**Proposition 7 (External consistency).** *If a client  $c$  executes “OAR-multicast( $m, \Pi$ )” (Fig. 5) and gets “reply =  $r$ ” (Fig. 5, line 5), then no correct processes ever*  
*(1) executes “reply  $\leftarrow$  A-deliver( $m$ )” (line 28) with “reply  $\neq r$ ”, or*  
*(2) executes “reply  $\leftarrow$  Opt-deliver( $m$ )” (line 26) with “reply  $\neq r$ ” without executing “Opt-undeliver( $m$ )” (line 26) during the same epoch.*

*Proof.* The client  $c$  waits for a set of replies that have a majority weight (Fig. 5, line 3) and selects one that has the highest individual weight (Fig. 5, lines 4–5). The replies sent by a server  $p$  can have two types of weight: a weight equal to  $\{p\}$  or  $\{p, s\}$  (optimistic weight), or a weight equal to  $\Pi$  (conservative weight). Optimistic weights are generated after  $p$  has Opt-delivered  $m$  (lines 13 and 15) and identify the processes that have Opt-delivered  $m$  according to its order in  $msgSet^k$  (line 11), defined by the sequencer during epoch  $k$ . Conservative weights are generated after  $p$  has A-delivered  $m$  (line 29) and indicate that all correct processes have agreed on the same conservative order (agreement property of *Cnsv-order*).

Therefore, we have only two cases to consider. The set of replies received by the client has a majority weight because (a) it contains at least one reply with a conservative weight, or (b) it contains replies that indicate that a majority of processes have Opt-delivered  $m$  according to the order defined by the sequencer during epoch  $k$ .

In case (a), let  $k$  be the epoch during which  $p$  has executed “reply  $\leftarrow$  A-deliver( $m$ )” with “reply =  $r$ ”. During epoch  $k$ , we have  $m \in New_p$  and the agreement property of *Cnsv-order* guarantees that for every server  $q$ , we have  $m \in (O\_delivered_q \ominus Bad_q) \oplus New_q$ . Therefore,  $q$  Opt-delivers (but does not Opt-undeliver) or A-delivers  $m$  during epoch  $k$ . By Proposition 5, either  $q$  executes “reply  $\leftarrow$  Opt-deliver( $m$ )” with “reply =  $r$ ” without executing “Opt-undeliver( $m$ )”, or  $q$  executes “reply  $\leftarrow$  A-deliver( $m$ )” with “reply =  $r$ ”.

In case (b), a majority of servers have Opt-delivered  $m$  according to the order sent by the sequencer in  $msgSet^k$ . Let  $p$  be some process that Opt-delivered  $m$ , and  $q$  some process that did not. Using the undo consistency property of *Cnsv-order*, we have  $m \notin Bad_p$  and  $m \notin Bad_q$ . Since  $m \in O\_delivered_p$ , by the agreement property of *Cnsv-order* we have  $m \in New_q$ . Therefore, all processes that Opt-delivered  $m$  will not Opt-undeliver it, and all processes that did not Opt-delivered  $m$  will A-deliver it. By Proposition 5,  $p$  has executed “reply  $\leftarrow$  Opt-deliver( $m$ )” with “reply =  $r$ ” and that  $q$  executes “reply  $\leftarrow$  A-deliver( $m$ )” with “reply =  $r$ ”.  $\square$

## A.3 Correctness of the *Cnsv-order* algorithm

In this section, we assume that all line numbers refer to Figure 7, unless specified otherwise.

**Lemma 2.** *Given any two sequences  $dlv_i$  and  $dlv_j$  in the decision  $D^k \equiv \{(dlv_1, notdlv_1); (dlv_2, notdlv_2); \dots\}$  of the consensus (line 4), if  $dlv_i$  and  $dlv_j$  are not equal, then one is a prefix of the other.*

*Proof.* By the maj-validity property of consensus, each  $dlv_i$  in  $D^k$  is the sequence of messages optimistically delivered by some process  $p$  during epoch  $k$ , i.e.,  $dlv_i = O\_delivered_p$ . Since, for each process  $p$ ,  $O\_delivered_p$  contains only messages ordered by the sequencer  $s$  (lines 8–10, Fig. 6) and communication channels are reliable and FIFO, each process receive the same sequence of  $msgSet^k$  messages from  $s$  (line 11, Fig. 6) and Opt-delivers messages in the same order. Therefore, for two processes  $p$  and  $q$ ,  $O\_delivered_p$  and  $O\_delivered_q$  can differ by their size (if  $s$  crashes while sending  $msgSet^k$  messages, only some servers may receive these messages), but one is a prefix of the other.  $\square$

**Lemma 3.** *The sequence  $dlv_{max}$  (line 5) is the same for all correct processes. The same holds for the sequence  $notdlv$  (line 13).*

*Proof.* The agreement property of consensus guarantees that all correct processes agree on the same decision  $D^k$ . Since (1) each process selects the longest  $dlv_i$  in  $D^k$  and (2) two sequences  $dlv_i$  and  $dlv_j$  in  $D^k$  with the same length are identical (Lemma 2), all processes select the same value for  $dlv_{max}$ .

At line 12, all processes deterministically assign  $notdlv$  using the  $notdlv_i$  sequences from  $D^k$ . Since  $D^k$  is identical for all correct processes,  $notdlv$  at line 12 is also identical for all correct processes. By line 13, and the equality of  $dlv_{max}$ , we have the equality of  $notdlv$ .  $\square$

**Lemma 4.** *For all correct processes  $p$ , after line 11, we have  $(O\_delivered_p \ominus Bad_p) \oplus New_p = dlv_{max}$ , where  $dlv_{max}$  is the longest sequence  $dlv_i$  contained in  $D^k$ .*

*Proof.* At line 6, process  $p$  check if  $O\_delivered_p$  is a prefix of  $dlv_{max}$ . If this is the case,  $p$  sets  $New_p$  to the messages from  $dlv_{max}$  that are not in  $O\_delivered_p$ , and  $Good_p$  to  $O\_delivered_p$  (lines 7–8). Therefore, since  $Bad_p$  is empty, we have  $Good_p = (O\_delivered_p \ominus Bad_p)$  and  $(O\_delivered_p \ominus Bad_p) \oplus New_p = dlv_{max}$ . If  $O\_delivered_p$  is not a prefix of  $dlv_{max}$ ,  $p$  sets  $Good_p$  to the common prefix of  $O\_delivered_p$  and  $dlv_{max}$ , and  $Bad_p$  to the messages from  $O\_delivered_p$  that are not in that prefix (lines 10–11). Therefore, since  $New_p$  is empty, as before we also have  $Good_p = (O\_delivered_p \ominus Bad_p)$  and  $(O\_delivered_p \ominus Bad_p) \oplus New_p = dlv_{max}$ .  $\square$

**Lemma 5.** *For all processes  $p$ , lines 15–19 do not modify the value of the expression  $(O\_delivered_p \ominus Bad_p) \oplus New_p$ .*

*Proof.* Immediate from the code.  $\square$

**Proposition 8 (Termination).** *If a correct process  $p$  calls  $Cnsv\_order$ , then  $p$  eventually gets the result  $\{Bad_p; New_p\}$ .*

*Proof.* Follows directly from the algorithm and the termination property of consensus.  $\square$

**Proposition 9 (Agreement).** *For any two correct processes  $p$  and  $q$ ,  $(O\_delivered_p \ominus Bad_p) \oplus New_p = (O\_delivered_q \ominus Bad_q) \oplus New_q$ .*

*Proof.* The agreement property of consensus ensures that all correct processes agree on the same decision  $D^k$ . At line 5, all processes  $p$  have the same value for  $dlv_{max}$  (Lemma 3). By Lemma 4, we have  $(O\_delivered_p \ominus Bad_p) \oplus New_p = dlv_{max}$  for all  $p$ , and thus the result holds after line 11. At line 12, all processes  $p$  generate the same value for  $notdlv$  (Lemma 3), and add  $notdlv$  to  $New_p$  (line 14). We now have  $(O\_delivered_p \ominus Bad_p) \oplus New_p = dlv_{max} \oplus notdlv$ . Thus, after line 14 (Fig. 7), the result still holds. By Lemma 5, the result is still true at the end of  $Cnsv\_order$ .  $\square$

**Proposition 10 (Unicity).** *For all processes  $p$ , we have  $New_p \cap (O\_delivered_p \ominus Bad_p) = \emptyset$ .*

*Proof.* We proof that, if a message  $m$  is part of  $O\_delivered_p$  but not of  $Bad_p$ , then it cannot be part of  $New_p$ . There are two places where a message  $m$  can have been added to  $New_p$ : at line 7 and at line 14.

At line 7, the sequence  $(dlv_{max} \ominus O\_delivered_p)$  is added to  $New_p$ . It follows immediately that, if  $m \in O\_delivered_p$ , we have  $m \notin New_p$ .

At line 14, the sequence  $(notdlv \ominus dlv_{max})$  is added to  $New_p$ . If  $m \in (O\_delivered_p \ominus Bad_p)$ , we have  $m \in dlv_{max}$  (Lemma 4) and thus  $m \notin New_p$ .  $\square$

**Proposition 11 (Non-triviality).** *If for a majority of processes  $q$ ,  $m \in O\_delivered_q \cup O\_notdelivered_q$ , then for all correct processes  $p$ , we have  $m \in (O\_delivered_p \ominus Bad_p) \oplus New_p$ .*

*Proof.* If for a majority of processes  $q$ ,  $m \in O\_delivered_q \cup O\_notdelivered_q$ , then for at least one pair of sequences  $\{O\_dlv_i; O\_notdlv_i\}$  of  $D^k$  we have either  $m \in O\_dlv_i$  or  $m \in O\_notdlv_i$ . If  $m \in O\_dlv_i$ , then by Lemma 2,  $m \in dlv_{max}$  at line 5. If  $m \in O\_notdlv_i$ , then  $m \in notdlv$  at line 12 (definition of  $\oplus$ ). So, at the end of *Cnsv-order*,  $m \in dlv_{max}$  or  $m \in notdlv$ . By Lemmata 4 and 5, we have  $m \in (O\_delivered_p \ominus Bad_p) \oplus New_p$ .  $\square$

**Proposition 12 (Validity).** *If for any process  $p$ ,  $m \in New_p$ , then for at least one process  $q$ , we have  $m \in O\_delivered_q \cup O\_notdelivered_q$ .*

*Proof.*  $New_p$  can only contain messages from  $dlv_{max}$  (line 7) or from  $notdlv$  (line 13). In the first case,  $m$  must be in the longest sequence  $O\_dlv_i$  of  $D^k$ , and in the second case  $m$  must be part of at least one sequence  $O\_notdlv_i$  of  $D^k$  (definition of  $\oplus$ ). In both cases, there must be a process  $q$  such that  $m \in O\_delivered_q \cup O\_notdelivered_q$ .  $\square$

**Proposition 13 (Undo legality).** *For all processes  $p$ , we have  $(O\_delivered_p \ominus Bad_p) \oplus Bad_p = O\_delivered_p$ .*

*Proof.* If  $Bad_p$  is empty, the proposition trivially holds.  $Bad_p$  takes a non empty value at lines 9–11, i.e., when  $O\_delivered_p$  is not a prefix of  $dlv_{max}$ . In this case  $dlv_{max}$  is a prefix of  $O\_delivered_p$  (Lemma 2), and  $Bad_p$  is set to  $O\_delivered_p \ominus dlv_{max}$  (line 11). It follows immediately that  $Bad_p$  is a suffix of  $O\_delivered_p$  after line 11. Since line 18 can only modify  $Bad_p$  by removing messages from the head of  $Bad_p$ , we conclude that  $Bad_p$  is still a suffix of  $O\_delivered_p$  at the end of *Cnsv-order*.  $\square$

**Proposition 14 (Undo consistency).** *For all processes  $p$ , if  $m \in Bad_p$  then for a majority of processes  $q$ , we have  $m \notin O\_delivered_q$ .*

*Proof.* A message  $m$  is in  $Bad_p$  only if  $m \in O\_delivered_p$  and  $m \notin dlv_{max}$ . If  $m \notin dlv_{max}$ , then  $m$  is not in any  $dlv_i$  of  $D^k$  (Lemma 2). Since for a majority of processes  $q$ ,  $D^k$  contains the sequence  $dlv_q$ , we also have  $m \notin O\_delivered_q$ .  $\square$

**Proposition 15 (Undo thriftiness).** *For all processes  $p$ , we have  $\odot(Bad_p, New_p) = \epsilon$ .*

*Proof.* The result follows immediately from lines 15–19 of the algorithm.  $\square$

**Proposition 16.** *The algorithm of Figure 7 satisfies the specification of *Cnsv-order* and is thrifty.*

*Proof.* Immediate from Propositions 8 to 15.  $\square$